

# BHARTIYA VIDYA BHAVAN’S

**SARDAR PATEL INSTITUTE OF TECHNOLOGY**

**Bhavan’s Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India**

NAME : Dhananjay Salunke

UID : 2020401073

CLASS : T.E.

BRANCH : I.T.

ACADEMIC YEAR : 2021-22 LAB :

Experiment 1 & 2

Traditional Crypto Methods and Key Exchange

**Objective:**

This experiment will be in two parts:

1. To implement Substitution, ROT 13, Transposition, Double Transposition, and Vernam Cipher in Scilab/C/Python/R.
2. Implement Diffie Hellman key exchange algorithm in Scilab/C/Python/R.



**Code:**

**Exp1 Code**

import string

def SubCypher():

# A list containing all characters

all\_letters= string.ascii\_letters

"""

create a dictionary to store the substitution

for the given alphabet in the plain text

based on the key

"""

dict1 = {}

plain\_txt= input("Enter Plain text to be encrypted: ")

key = int(input("Enter no. of position shifts: "))

for i in range(len(all\_letters)):

dict1[all\_letters[i]] = all\_letters[(i+key)%len(all\_letters)]

cipher\_txt=[]

# loop to generate ciphertext

for char in plain\_txt:

if char in all\_letters:

temp = dict1[char]

cipher\_txt.append(temp)

else:

temp =char

cipher\_txt.append(temp)

cipher\_txt= "".join(cipher\_txt)

print("Encrypted Message: ",cipher\_txt)

#create a dictionary to store the substitution

#for the given alphabet in the cipher

#text based on the key

dict2 = {}

for i in range(len(all\_letters)):

dict2[all\_letters[i]] = all\_letters[(i-key)%(len(all\_letters))]

# loop to recover plain text

decrypt\_txt = []

for char in cipher\_txt:

if char in all\_letters:

temp = dict2[char]

decrypt\_txt.append(temp)

else:

temp = char

decrypt\_txt.append(temp)

decrypt\_txt = "".join(decrypt\_txt)

print("Decrypted Message: ", decrypt\_txt)

def ROT13():

# Dictionary to lookup the index of alphabets

dict1 = {'A' : 1, 'B' : 2, 'C' : 3, 'D' : 4, 'E' : 5,

'F' : 6, 'G' : 7, 'H' : 8, 'I' : 9, 'J' : 10,

'K' : 11, 'L' : 12, 'M' : 13, 'N' : 14, 'O' : 15,

'P' : 16, 'Q' : 17, 'R' : 18, 'S' : 19, 'T' : 20,

'U' : 21, 'V' : 22, 'W' : 23, 'X' : 24, 'Y' : 25, 'Z' : 26}

# Dictionary to lookup alphabets

# corresponding to the index after shift

dict2 = {0 : 'Z', 1 : 'A', 2 : 'B', 3 : 'C', 4 : 'D', 5 : 'E',

6 : 'F', 7 : 'G', 8 : 'H', 9 : 'I', 10 : 'J',

11 : 'K', 12 : 'L', 13 : 'M', 14 : 'N', 15 : 'O',

16 : 'P', 17 : 'Q', 18 : 'R', 19 : 'S', 20 : 'T',

21 : 'U', 22 : 'V', 23 : 'W', 24 : 'X', 25 : 'Y'}

# Function to encrypt the string

# according to the shift provided

def encrypt(message, shift):

cipher = ''

for letter in message:

# checking for space

if(letter != ' '):

# looks up the dictionary and

# adds the shift to the index

num = ( dict1[letter] + shift ) % 26

# looks up the second dictionary for

# the shifted alphabets and adds them

cipher += dict2[num]

else:

# adds space

cipher += ' '

return cipher

# Function to decrypt the string

# according to the shift provided

def decrypt(message, shift):

decipher = ''

for letter in message:

# checks for space

if(letter != ' '):

# looks up the dictionary and

# subtracts the shift to the index

num = ( dict1[letter] - shift + 26) % 26

# looks up the second dictionary for the

# shifted alphabets and adds them

decipher += dict2[num]

else:

# adds space

decipher += ' '

return decipher

# function to run the program

def main():

# use 'upper()' function to convert any lowercase characters to uppercase

message = input("Enter plain text to be encrypted: ")

shift = 13

result = encrypt(message.upper(), shift)

print("Encrypted Cypher: ", result)

message = result

shift = 13

result = decrypt(message.upper(), shift)

print ("Decrypted Cypher: ",result)

# Executes the main function

if \_\_name\_\_ == '\_\_main\_\_':

main()

def TranspoCypher():

import math

def encryptMessage(msg):

cipher = ""

# track key indices

k\_indx = 0

msg\_len = float(len(msg))

msg\_lst = list(msg)

key\_lst = sorted(list(key))

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# add the padding character '\_' in empty

# the empty cell of the matix

fill\_null = int((row \* col) - msg\_len)

msg\_lst.extend('\_' \* fill\_null)

# create Matrix and insert message and

# padding characters row-wise

matrix = [msg\_lst[i: i + col]

for i in range(0, len(msg\_lst), col)]

# read matrix column-wise using key

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

cipher += ''.join([row[curr\_idx]

for row in matrix])

k\_indx += 1

return cipher

def decryptMessage(cipher):

msg = ""

# track key indices

k\_indx = 0

# track msg indices

msg\_indx = 0

msg\_len = float(len(cipher))

msg\_lst = list(cipher)

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# convert key into list and sort

# alphabetically so we can access

# each character by its alphabetical position.

key\_lst = sorted(list(key))

# create an empty matrix to

# store deciphered message

dec\_cipher = []

for \_ in range(row):

dec\_cipher += [[None] \* col]

# Arrange the matrix column wise according

# to permutation order by adding into new matrix

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

for j in range(row):

dec\_cipher[j][curr\_idx] = msg\_lst[msg\_indx]

msg\_indx += 1

k\_indx += 1

# convert decrypted msg matrix into a string

try:

msg = ''.join(sum(dec\_cipher, []))

except TypeError:

raise TypeError("This program cannot",

"handle repeating words.")

null\_count = msg.count('\_')

if null\_count > 0:

return msg[: -null\_count]

return msg

msg = input("Enter Plain text to be Encrypted: ")

key = input("Enter Key: ")

cipher = encryptMessage(msg)

print("Encrypted Message: {}".

format(cipher))

print("Decryped Message: {}".

format(decryptMessage(cipher)))

def DoubTranspoCypher():

import math

def encryptMessage(msg, key):

cipher = ""

# track key indices

k\_indx = 0

msg\_len = float(len(msg))

msg\_lst = list(msg)

key\_lst = sorted(list(key))

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# add the padding character '\_' in

# the empty cell of the matix

fill\_null = int((row \* col) - msg\_len)

msg\_lst.extend('\_' \* fill\_null)

# create Matrix and insert message and

# padding characters row-wise

matrix = [msg\_lst[i: i + col]

for i in range(0, len(msg\_lst), col)]

# read matrix column-wise using key

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

cipher += ''.join([row[curr\_idx]

for row in matrix])

k\_indx += 1

return cipher

def decryptMessage(cipher, key):

msg = ""

# track key indices

k\_indx = 0

# track msg indices

msg\_indx = 0

msg\_len = float(len(cipher))

msg\_lst = list(cipher)

# calculate column of the matrix

col = len(key)

# calculate maximum row of the matrix

row = int(math.ceil(msg\_len / col))

# convert key into list and sort

# alphabetically so we can access

# each character by its alphabetical position.

key\_lst = sorted(list(key))

# create an empty matrix to

# store deciphered message

dec\_cipher = []

for \_ in range(row):

dec\_cipher += [[None] \* col]

# Arrange the matrix column wise according

# to permutation order by adding into new matrix

for \_ in range(col):

curr\_idx = key.index(key\_lst[k\_indx])

for j in range(row):

dec\_cipher[j][curr\_idx] = msg\_lst[msg\_indx]

msg\_indx += 1

k\_indx += 1

# convert decrypted msg matrix into a string

try:

msg = ''.join(sum(dec\_cipher, []))

except TypeError:

raise TypeError("This program cannot",

"handle repeating words.")

"""

if null\_count > 0:

return msg[: -null\_count]

"""

return msg

# Driver Code

msg = input("\nEnter Message to be Encrypted: ")

key1 = input("key 1: ")

key2 = input("key 2: ")

cipher = encryptMessage(msg, key1)

print("\nEncrypted Message with key 1: {}".format(cipher))

cipher = encryptMessage(cipher, key2)

print("\nEncrypted Message with key 2: {}".format(cipher))

print("\nDecrypted Message with key 2: {}".format(decryptMessage(cipher, key2)))

decryCypherMess = decryptMessage(cipher, key2)

print("\nDecrypted Message with key 1: {}".format(

decryptMessage(decryCypherMess, key1)))

def VernamCypher():

def VernamCipherFunction(text, key):

result = "";

ptr = 0;

for char in text:

result = result + chr(ord(char) ^ ord(key[ptr]));

ptr = ptr + 1;

if ptr == len(key):

ptr = 0;

return result

input\_text = input("\nEnter Text To Encrypt:\t");

encryption\_key = input("Input key: ");

encryption = VernamCipherFunction(input\_text, encryption\_key);

print("\nEncrypted Vernam Cipher Text:\t" + encryption);

decryption = VernamCipherFunction(encryption, encryption\_key);

print("\nDecrypted Vernam Cipher Text:\t" + decryption);

def main():

flag=1;

while flag==1:

print("\n\t\t\*\*\*\*\*\* -------------------------\*\*\*\*\*\*\t\t");

print("\n\n Your Options\n");

print("1. Substitution Cypher");

print("2. ROT13 Cypher");

print("3. Transposition Cypher");

print("4. Double Transposition Cypher");

print("5. Vernam Cypher");

print("0. Exit");

c=int(input("Enter your choice:"));

if c==1:

print("\n\t\t\*\*\*\*\*\* Substitution Cypher \*\*\*\*\*\*\t\t");

SubCypher();

elif c==2:

print("\n\t\t\*\*\*\*\*\* ROT13 Cypher \*\*\*\*\*\*\t\t");

ROT13();

elif c==3:

print("\n\t\t\*\*\*\*\*\* Transposition Cypher \*\*\*\*\*\*\t\t");

TranspoCypher();

elif c==4:

print("\n\t\t\*\*\*\*\*\* Double Transposition Cypher \*\*\*\*\*\*\t\t");

DoubTranspoCypher();

elif c==5:

print("\n\t\t\*\*\*\*\*\* Vernam Cypher \*\*\*\*\*\*\t\t");

VernamCypher();

elif c==0:

print("\n\t\t\*\*\*\*\*\* EXIT \*\*\*\*\*\*\t\t");

flag=0;

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Exp2 Code**

#diffie hel Lman

g = int(input("Enter the first prime number : "))

n = int(input("Enter the second prime number : "))

x = int(input("Enter the secret X for Alice : "))

y = int(input("Enter the secret Y for Bob : "))

public1 = (n\*\*x)%g

public2 = (n\*\*y)%g

print("The generated key of Alice is : ",public1)

print("The generated key of Bob is : ",public2)

secret1 = (public1\*\*y)%g

secret2 = (public2\*\*x)%g

print("Their shared secret is : ",secret1)

print("Their shared secret is : ",secret

**Explanation:**

**Substitution Cipher:**

* Create A list containing all characters
* Create a dictionary to store the substitution for the given alphabet in the plain text based on
* the key
* Loop to generate ciphertext
* Print Cypher Text(Encrypted Message)
* create a dictionary to store the substitution for the given alphabet in the cipher text based
* on the key
* loop to recover plain text
* print Decrypted Message

**ROT13 Cipher:**

* This script uses dictionaries instead of 'chr()' & 'ord()' function.
* Create a Dictionary to lookup the index of alphabets
* Create Dictionary to lookup alphabets corresponding to the index after shift
* Function to encrypt the string according to the shift provided
* Check For spaces
* Looks up the dictionary and adds the shift to the index
* Looks up the second dictionary for the shifted alphabets and adds them
* Return Cipher
* Function to decrypt the strings according to the shift provided
* checks for space
* Looks up the dictionary and subtracts the shift to the index
* Looks up the second dictionary for the shifted alphabets and adds them
* adds space
* Driver Function
* use 'upper()' function to convert any lowercase characters to uppercase
* take inputs and call encrypt and decrypt application

**Transposition Cipher:**

 # Encryption Function

* Track key indices using k\_indx = 0
* Calculate column of the matrix
* Calculate maximum row of the matrix
* add the padding character '\_' in empty the empty cell of the matix
* create Matrix and insert message and padding characters row-wise
* read matrix column-wise using key

# Decryption Function

* Track key indices
* Track msg indices
* calculate column of the matrix
* calculate maximum row of the matrix
* convert key into list and sort alphabetically so we can access each character by its
* alphabetical position.
* create an empty matrix to store deciphered message
* Arrange the matrix column wise according to permutation order by adding into new
* matrix
* Convert decrypted msg matrix into a string

Driver Function

* Input message to be encrypted and key.
* Calling Respective Functions for Encryption and Decryption

**Double Transposition Cipher:**

# Encryption Function

* Track key indices using k\_indx = 0
* Calculate column of the matrix
* Calculate maximum row of the matrix
* add the padding character '\_' in empty the empty cell of the matix
* create Matrix and insert message and padding characters row-wise
* read matrix column-wise using key

# Decryption Function

* Track key indices
* Track msg indices
* calculate column of the matrix
* calculate maximum row of the matrix
* convert key into list and sort alphabetically so we can access each character by its
* alphabetical position.
* create an empty matrix to store deciphered message
* Arrange the matrix column wise according to permutation order by adding into new
* matrix
* Convert decrypted msg matrix into a string

Driver Function

* Input message to be encrypted , key 1 and key 2.
* Calling Respective Functions for Encryption and Decryption twice respectively.

**Vernam Cipher:**

* For Encryption and Decryption we are using same function
* Encryption/Decryption Function
* Create empty Result string
* Fill this Result string by XORing with key using for loop
* Return results

Driver Function

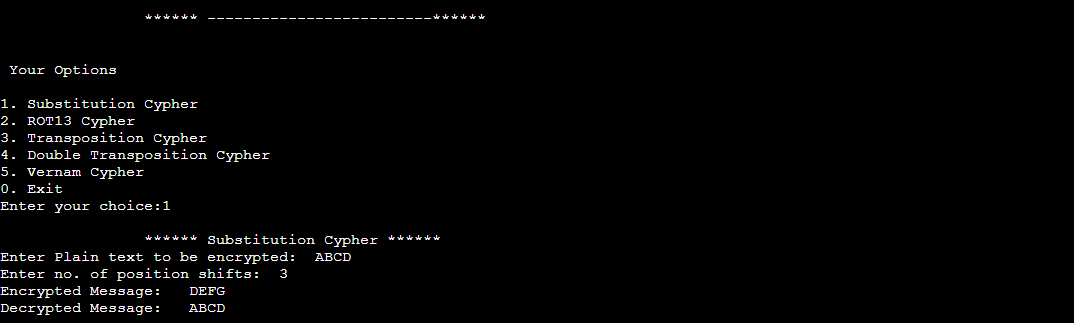
* Input message to be encrypted and encryption key.
* Calling Encryption and Decryption Function Respectively.

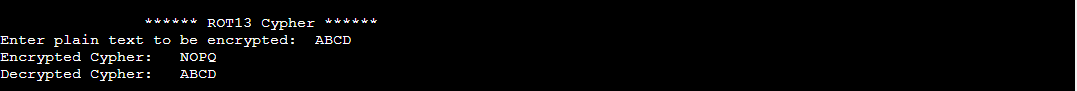
**Defie-Hellman:**

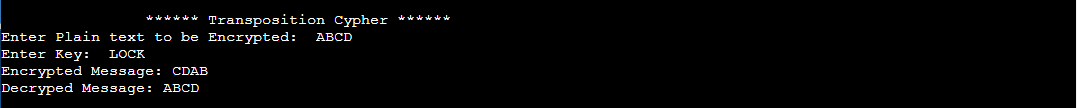
* Alice and Bob get public numbers P = 23, G = 9
* 
* Alice selected a private key a = 4 and
* Bob selected a private key b = 3
* 
* Alice and Bob compute public values
* x = (9^4 mod 23) = (6561 mod 23) = 6
* y = (9^3 mod 23) = (729 mod 23) = 16
* 
* Alice and Bob exchange public numbers
* 
* Alice receives public key y =16 and
* Bob receives public key x = 6
* 
* Alice and Bob compute symmetric keys
* Alice: ka = y^a mod p = 65536 mod 23 = 9
* Bob: kb = x^b mod p = 216 mod 23 = 9
* 
* 9 is the shared secret.

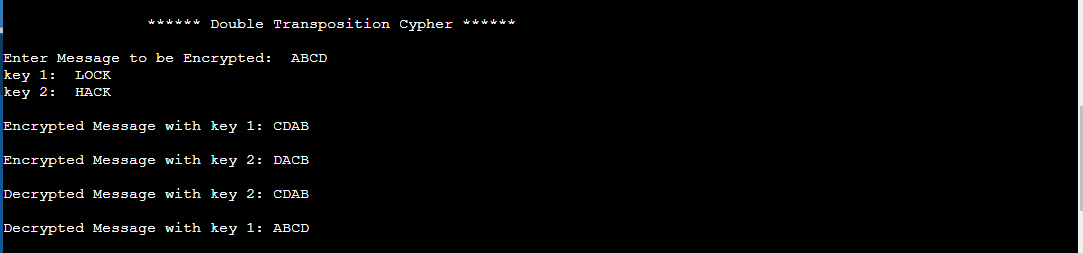


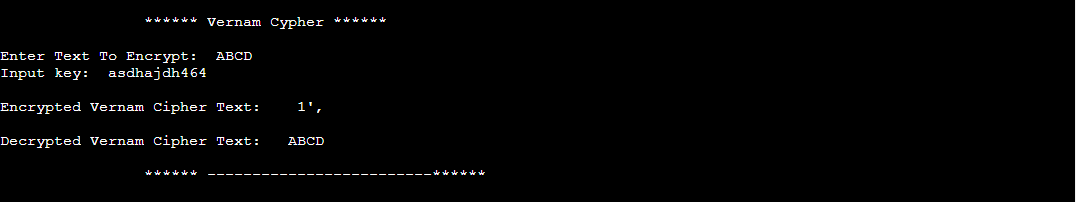
**Output:**

****

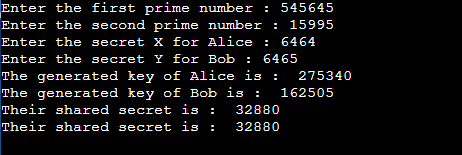
****







Exp 2



# Observations and Conclusion:



* **Substitution Cipher** is very easy to decipher it is just shifting the alphabets in order to encrypt and decrypt a plaintext letter. It is very basic encryption cipher.
* **ROT 13** is also very simple to crack as it is substitution cipher shifted by 13, also we need to use modulo operations in order to keep it going circular otherwise we mess up in capital and small letters. Also here small letters are not allowed only capital letters are allowed in text as I have mapped capital letters separately to numbers.
* In **transposition cipher** the message is written out in rows of a fixed length, and then read out again column by column and placed in matrix, and the columns are chosen in some unknown random order. More importantly Keyword defines width and Permutation of columns. Any spare spaces are filled with nulls or left blank or placed by a character to keep matrix consistent (Example: \_). In order to decipher it, the user has to work out the **(column lengths by dividing the message length by the key length)**. Then they can write the message out in columns again, then re-order the columns by reforming the key word.

# What lead to discovery of Double Transposition cipher?

* + If multiple messages of exactly the same length are encrypted using the same keys, they can be anagrammed simultaneously. By mapping frequently used words like “THE”, “OF”, etc. We can identify arrangement of columns. This can lead to both recovery of the messages, and to recovery of the keys (so that every other message sent with those keys can be read).
* **Double transposition cipher** offers better security than the single Transposition cipher, it is just 2 times encryption and decryption using single transposition cipher. With two same or different keys.
* **The Vernam cipher** : In this cipher instead of a single key, each plaintext character is encrypted using its own key. This key or key stream is taken from a **one-time pad**. The key must be equal in length to the plaintext message. The fact that each character of the message is encrypted using a different key prevents any useful information being revealed through a frequency analysis of the ciphertext. To encrypt we have to XOR key with message. I got some surprising result when I worked with chr(ord(char) and XORed it. Although encryption appeared to be weird but decryption of that encrypted text was perfectly fine.
* **My conclusion about Diffie–Hellman key exchange** is it is a method of securely exchanging cryptographic keys over a public channel. It is a method
  + of digital encryption that uses numbers raised to specific powers to produce decryption keys on the basis of components that are never directly transmitted. Although it is not tough for someone snooping the network to decrypt the data and

get the keys, if the numbers generated are not entirely random. Hence it is recommended to keep larger and random keys.

